Project 3: CMUD

1   Overview

What you will learn:

- Collaboratively developing a large-scale system with multiple different components communicating via RPCs (Lecture Sep 27)

- Enforcing consistency guarantees for replicated data (Lecture Oct 6)

- Using an Actor Model to coordinate across cores and machines using message passing (Lecture Oct 27)

- Working with a scalable web service architecture (Lecture Nov 10)

Important dates:

Project release: **Friday, November 18, 2022**
Checkpoint due: **Thursday, December 1, 2022 at 11:59pm**
Final Project due: **Friday, December 9, 2022 at 11:59pm**

Logistics:

You must work on this project **with a partner who is also in the class**. As with P1, only one member of each group will need to submit to Gradescope (check the README for submission guidelines), and after submitting, make sure to add your partner under Group Members.

You will have **15 submissions** for each due date. Your Gradescope submission will output a message showing how many submissions you have made and how many you have left. Gradescope will allow you to submit beyond this limit, but we will be checking manually. **Only your activated submission counts** (by default your last submission).

Gradescope will count how many submissions are under your name, and help you calculate the number of remaining submissions. To better keep track of the number of submissions for your group, please add your partner as a Group Member for EACH submission. We won't accept requests to update scores if you miscount the number of submissions of your group.

The same late policy as previous projects will apply here as well—at most 2 late days are automatically granted for every assignment for **valid reasons** with no penalty. No submissions will be accepted 2 days after each deadline.

Please note that there is one checkpoint for this project, which is **25%** of the final grade. For more information regarding what portion of the project is expected to be completed for the checkpoint, please refer to Section 8.

The starter code for this project is hosted as a read-only repository on GitHub. For instructions on how to run, test, and submit your server implementation, see the README.md file in the project's root directory. To clone a copy, execute the following Git command:

```
git clone https://github.com/15-440/p3.git
```

Warning: do not fork the repo and then do your work in there, since forks of public repos on Github are public by default. If you want to manage your code using Github, instead clone the repo and then push the code to a private repo that is only accessible to you and your partner.

2   Architecture

In this project, you will implement portions of a multiplayer online game called *CMUD*. In CMUD, players progress through the ranks of CMU computer science by mining coins, improving their hardware, and solving hard problems. The game is a Multi-User Dungeon (MUD), meaning that all players inhabit the same virtual world and interact with it through a text terminal.

Unlike a traditional MUD with a single application server, CMUD uses the globally distributed architecture shown in Figure 1. Players run the CMUD client app on their own machine, which connects to a nearby (low ping) backend storage server. The CMUD client is stateless, converting the player's inputs into queries against the backend storage server and printing out the results. Different backend storage servers "sync" with each other (i.e., exchange data in the background) to maintain a single consistent global state, so that all users see the same virtual world, including each others' actions.

The backend storage system itself implements a simple *key-value store* for string keys and values, similar to Project 0. Specifically, it supports operations to Get and Put key value pairs, plus an operation to List all key-value pairs starting with a given prefix. Since this API is very simple, all application logic happens on the client, unlike the classic three-tier architecture with separate application and backend servers; see Figure 2.

The advantage of this architecture is that it simplifies app development and deployment: a cloud provider can run the same generic backend storage system for many apps, while individual app developers only need to code and deploy their app's client, not their own
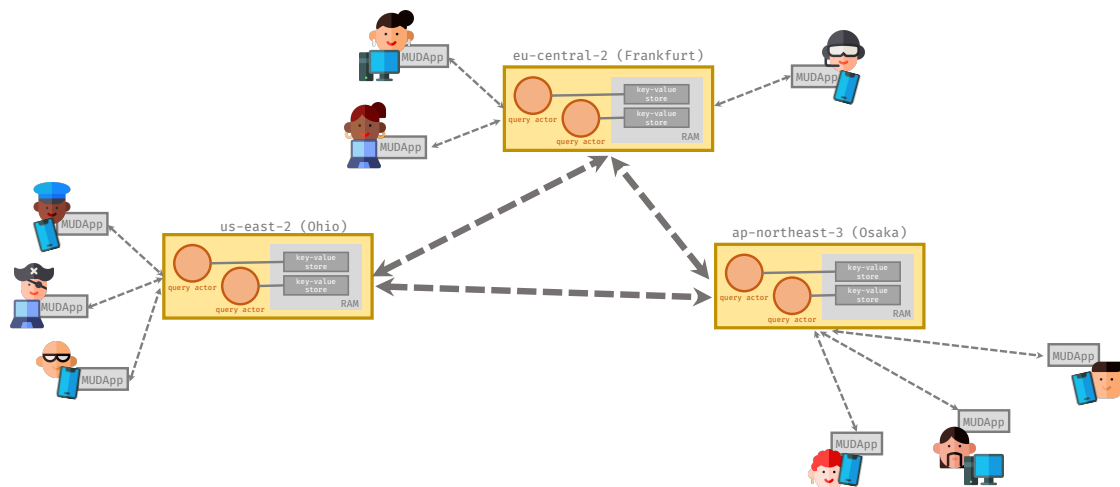
Figure 1: CMUD's globally distributed architecture.

servers.[1] In this project, though, you will be implementing the backend storage system, while we provide most of the CMUD client app as starter code.

2.1 Components

In more detail, CMUD's components are as follows.

CMUD App The CMUD app is the layer seen by the user. Its function is to interpret user commands, convert them into backend operations, and present the results back to the user. For this project, the CMUD-specific part of the client has been provided as starter code, located in the `app` package. However, you will have to implement:

Storage Client (Section 6) The storage client is the library that an app uses to interact with the backend storage system, located in the `kvclient` package. It converts Go function calls into RPCs for a nearby backend storage server.

Backend Storage System (Section 3) The backend storage system provides a key/value storage service for string values (much like a hash table), located in the `kvserver` package. Each storage server will support `Get`, `Put`, and `List` queries. In addition, the storage system will support the following features:

---

[1]This is sometimes called a "serverless" deployment model, since app developers do need to deploy or manage individual servers, although of course there are servers somewhere in the system. For a real-world example of a system that supports apps in this way, see Google's Cloud Firestore.
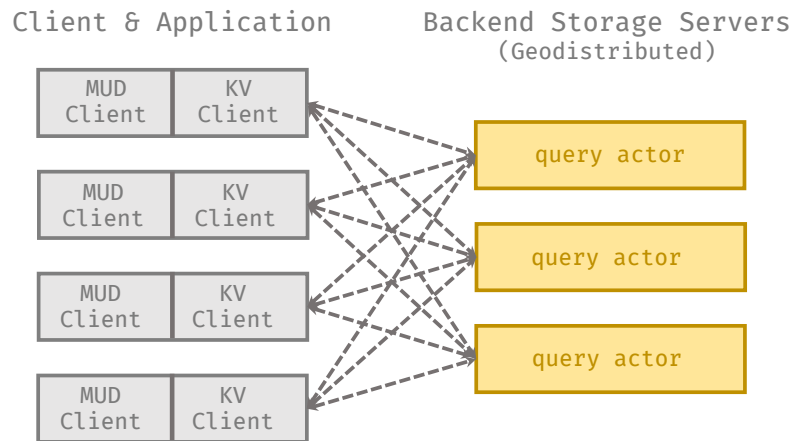
Figure 2: Layers in CMUD: clients (doubling as application layer) and backend storage servers.

- It provides low latency to all clients by being *geographically distributed* (many servers around the world, so each client is close to one) and *highly available* (each server can answer queries without waiting to coordinate with others).

- It uses multiple CPU cores per server to answer queries in parallel, increasing throughput.

- It enforces *eventual consistency* in the face of concurrent queries using a last-writer-wins rule.

Actor System (Section 4) The backend storage system is itself built on top of an **actor system**, located in the actor package. This means that it consists of independent processing units called *actors*, described in Section 4. The actor system is responsible for managing these actors and passing messages between them. Most of the actor package has been provided as starter code, but you will have to implement the pieces described in Section 4.2.

We begin by describing the backend storage system in detail.

3   Backend Storage System

In jargon, the backend storage system (package kvserver) is a "geographically distributed, highly available, NoSQL key-value store". NoSQL means that it provides data storage and retrieval mechanisms, but it does not use a relational model like traditional databases.

Instead, it is a key-value store with a simple Get/Put data model. The backend's design is based on Anna KVS [5], with additional inspiration from Apache Cassandra [1], Amazon Dynamo [3], and others [2].

The backend storage system consists of one or more servers (type `*kvserver.Server`), distributed across cloud or edge data centers around the world. Each server runs a single Go process that responds to key-value store queries on several ports—nominally one port per CPU core, although we may violate this for tests' sake.

For each port, the server runs a *query actor* (file `kvserver/query_actor.go`) that responds to queries sent to that port. Each query actor is an *actor*; we'll describe these in detail in Section 4, but for now, you can think of an actor like a thread or goroutine that is **not** allowed to share memory with other actors. Because of this "shared-nothing" constraint, each actor stores a replica of the entire key-value store in its own private hash table.

All query actors on all servers are replicas of the same data, so a storage system client can send queries to any query actor. Typically, a client will choose a random query actor (for load balancing) on a geographically nearby server (for low latency).

3.1   Queries

Supported queries on each port (equivalently, each query actor) are:

- `Get(key)`: Returns the value associated with `key`, if present.

- `Put(key, value)`: Sets the value associated with `key`.

- `List(prefix)`: Returns all `(key, value)` pairs whose key starts with `prefix`, similar to recursively listing all files in a folder.

RPCs for these queries are defined by the `QueryReceiver` interface in `kvcommon/rpc_types.go`. In `NewServer` (file `kvserver/server.go`), for each query actor, you should start an RPC server on the appropriate port and register a receiver implementing `QueryReceiver`. Each `QueryReceiver` RPC handler should forward its query to its actor (via `ActorSystem.Tell`), wait for a response from that actor (via `ActorSystem.NewChannelRef`), and then reply to its caller. See Figure 3.

Each query actor answers queries (forwarded by its `QueryReceiver`) using its local hash table. For `List`, it is okay for the actor to loop over its whole hash table checking for keys with the given prefix.[2]

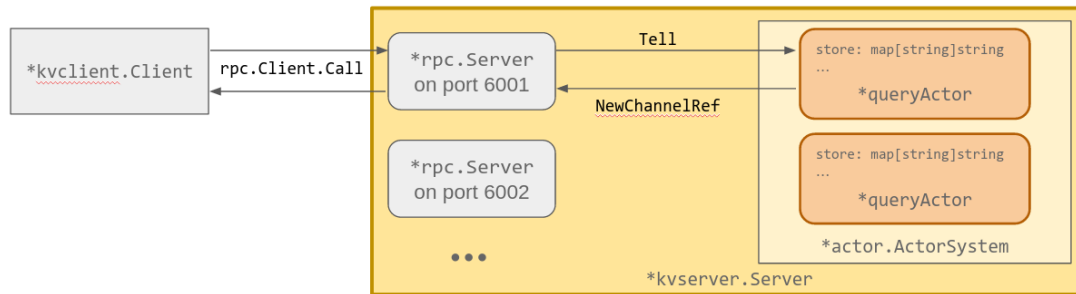[2] A real implementation would use a fancier data structure than a hash table to make `List` queries efficient.

Figure 3: Methods involved in answering a client query.

To get started with RPCs, please consult the net/rpc package docs and the commented sample code in `kvserver/server.go`. Keep in mind Go's rules around methods made available for remote access.

3.2   Consistency (Syncing)

To keep game latency low, query actors always reply to queries immediately, using their private hash table's current state. In particular, each query actor applies `Put` queries to its own hash table immediately and shows that value to future reads (`Get` and `List`).

To keep all query actors' states consistent, each query actor occasionally communicates with other query actors (both on the same server and on remote servers) to inform them of newly-`Put` values. We refer to this communication process as *syncing.*[3]

You will implement syncing by sending messages between actors using the actor system, as described in Section 5. As a prerequisite, though, we first describe actors and the actor system in detail.

4   Actor System

Recall from the Distributed Computing lecture (Oct 27) that an *actor* is an abstraction of a process that:

- Processes messages from a FIFO mailbox;

- May have local state, but may **not** share memory with others;

---

[3] Note that this is distinct from synchronization in the sense of e.g. mutexes.

- May send a point-to-point message to any other actor in its distributed system, asynchronously adding that message to the recipient's mailbox.

For example, the following shows pseudocode for a simple "counter actor":

```
actor state: count int

def actor.OnMessage(message):
    switch (message type):
        case MAdd:
            actor.count += message.Value
        case MGet:
            send MResult{actor.count} to message.Sender
```

This actor stores a count as its local state and receives `MAdd` and `MGet` messages. When another actor sends it an `MAdd` message, it adds the value indicated in the message to its count. When another actor sends it an `MGet` message, it responds with its current value by communicating in the only way it can: sending a message, in this case back to the sender, which is identified by an *actor ref* included in the message.

Observe that `OnMessage` proceeds sequentially: it processes a message step-by-step and returns, ready to process the next message, without syncing or waiting for responses. In particular, sending a message to another actor never blocks; the sent message is added to that actor's mailbox, so sending proceeds even if the other actor is busy processing prior messages.

In the actor system you will use, messages have the following delivery guarantees, similar to Akka's:

- Messages are delivered in order *per sender-receiver pair*.

- Messages are delivered *at-most-once*, so message delivery may fail. Failures are silent, i.e., the sender is not explicitly notified.

- Generally, messages will be delivered exactly-once unless someone goes wrong (e.g., a network or server failure). **For this project, you do not need to handle server or network failures, so in particular you do not need to resend dropped messages.**

The actor programming model allows coordination both within the same machine and across the network, without blocking, locking, or race conditions. Because actors cannot share memory, you need to architect actor programs differently than you are used to, but

this often leads to programs that are easier to distribute across multiple servers. Actors also have fault-tolerance benefits, but because you do not need to handle failures for this project, the provided `actor` package omits fault-tolerance features.

4.1 Package `actor`[4]

Package `actor` provides `ActorSystem` (file `actor/actor_system.go`), a basic Go *actor system* inspired by Akka. `ActorSystem` is responsible for starting and running actors, delivering messages to actors' mailboxes, and sending and receiving messages to/from actors in remote `ActorSystem`s. It also has functions to communicate with actors from outside the actor system. Typically, you will have one `ActorSystem` instance per process, running all actors in that process.

Actors are implementers of the `Actor` interface (file `actor/actor.go`), which has a single function `OnMessage(message interface) error`. This function is called in-order for each message sent to the actor (if the actor falls behind, messages are buffered FIFO in its mailbox). Inside `OnMessage`, the actor may read and write its local state (i.e., its `struct` fields) and send messages to other actors.

4.2 Finishing the Actor System

You will need to implement two small pieces of the actor system:

1. **For the checkpoint**, fill in the `Mailbox` implementation in `actor/mailbox.go`. This is the FIFO mailbox that `ActorSystem` uses to queue messages for delivery to a particular actor or remote server. It essentially behaves like an infinite-buffered Go channel. See the file's doc comments for more details.

2. **For the final deadline**, fill in the functions in `actor/remote_tell.go`. These are used to deliver messages from one `ActorSystem` to a remote `ActorSystem`. Internally, they should use RPCs.

4.3 Using the `actor` Package

We now describe how to use the `actor` package, including some important restrictions on actors. Full info can be found in the `actor` package documentation. We also provide an

---

[4]On a first reading, you may wish to skip to Section 5, which describes how you will actually use actors in your backend storage system.

example in the `example` folder; you are free to copy and modify it. Note that the `example` code will only work after you finish the `Mailbox` implementation.

First, import the package as `"github.com/cmu440/actor"`. Next, call

```
system, err := actor.NewActorSystem(port)
```

to get an instance of `*ActorSystem`. The `port` is used to listen for messages from remote actors running in remote `ActorSystem`s. For debugging purposes, you'll probably want to register an error handler via `system.OnError`.

Once you have an `ActorSystem` and an `Actor` implementation, such as the `counterActor` in `example/counter_actor.go` (which implements the pseudocode actor above), you can create instances with

```
ref := system.StartActor(newCounterActor)
```

Here `newCounterActor` is a "constructor" that returns a new `counterActor` instance (as an `Actor` implementation), defined in `example/counter_actor.go`.

`system.StartActor` returns an `*actor.ActorRef`. You can use this actor ref to send messages to your new actor instance from outside the actor system:

```
system.Tell(ref, MAdd{1})
```

You can also include `ref` in messages that you send to other actors, so that those actors can send messages to this one.

Finally, to do anything interesting with your actors, you'll need a way to get information back out of the actor system. To do that, see `ActorSystem.NewChannelRef` (example in `example/main.go`).

4.4   Restrictions

Your `Actor` implementations may not coordinate with other parts of the distributed system in any way besides receiving and sending messages. In particular, actors may not access shared memory, use Go channels, spawn goroutines, perform I/O (except logging), or block.

To help enforce this, the `ActorSystem` will marshal and unmarshal messages sent between actors, as if they are RPC call args, even when the source and destination are actors on the same machine. To make this possible, **each actor message type must be marshallable with Go's encoding/gob package.** In particular:

9

- All struct types, struct fields, and nested struct fields must be exported (Capitalized).

- Messages must not contain Go channels, functions, or similar non-marshallable types. Pointers are okay, but the contents they point to will be copied, not shared as a pointer.

- We recommend defining message types as plain structs, not pointers to structs.

- Any message type that is a struct must be registered with gob.Register. We recommend doing this in an `init` function in the same file where your actor is defined (example in `example/counter_actor.go`).

Also, we force you to create actors in a "constructor" with a fixed signature—namely, the argument to `ActorSystem.StartActor`. To pass initial data to an actor, instead of changing the values you set in the constructor, use the `ActorSystem` to send it a message.

Please do not circumvent these safeguards, e.g., by accessing mutable global variables or closure variables inside an actor or its constructor. We will deduct substantial points during manual grading if we notice this.

5   More on Consistency (Syncing)

Recall from Section 3 that all query actors are replicas of the same data: they each keep their own private copy of the key-value store, and these stores are supposed to remain consistent with each other. To do this, query actors need to sync with each other, i.e., communicate newly-`Put` values to each other.

To permit high availability (query actors accept `Puts` without coordination), this syncing happens in the background, off the critical path of responding to queries. As usual for actors, syncing happens by sending messages between actors. The exact sync strategy (i.e., what messages, to/from which actors, and how often) is up to you, subject to the requirements below. For manual grading, **please document your sync strategy (local & remote) with a 1-2 paragraph comment in** `kvserver/query_actor.go`.

Due to the highly available design, it is possible for multiple query actors to receive `Puts` for the same key concurrently. That will put them in a temporarily inconsistent state. We permit this so long as they are *eventually consistent* [4]: all query actors eventually see all updates, and once two actors have seen the same set of updates, they have the same state.[5]

---

[5]This is an even weaker consistency requirement than causal consistency, the weakest consistency guarantee described in the Distributed Replication lecture (Oct 6). It is possible to implement causal consistency in a highly available setting, but somewhat more complicated and expensive.

Specifically, you will implement a *last-writer-wins (LWW)* rule for `Puts`. For each `Put(key)` operation, attach a local wall-clock time on the server side, with millisecond resolution. Upon receiving a `Put(key)` from another actor, compare its timestamp to that of your current state for `key` (i.e., corresponding to the `Put(key)` query that set the current value for `key`), and ignore if it is lesser. Break ties arbitrarily but deterministically, e.g., using a lexicographic sort on the `ActorRef.Uid()` of the query actor who processed the original `Put`. We refer to this receiving logic as the *merge function*, since it merges a received `Put` into your local state.

Last-writer-wins ensures that you always see the chronologically last write to a key.[6] Also, the merge function is Associative, Commutative, and Idempotent (ACI): merging updates in any order, with any number of duplicates, always gives the same result—namely, the result of the `Put` with the greatest timestamp. This ensures eventual consistency even if different actors receive updates in different orders or receive duplicate updates.

5.1   Local Syncing

We refer to different query actors running on the same server as *local actors*. You will need to implement *local syncing* to ensure that these actors' states are eventually consistent, following the last-writer-wins rule above.

You will find it necessary to inform local actors of each others' existence in `NewServer`, so that they can send messages to each other. To do this, you can use `ActorSystem.Tell` to pass an initial message to each actor containing `*actor.ActorRef`s for other local actors.

Other requirements:

- `Puts` on one actor should be visible to `Get` and `List` queries on other local actors within **500 ms**.

- To amortize overhead, for each ordered pair of actors (A, B), A should send **no more than 10 messages per second** to B, regardless of how many `Puts` A receives.

- For efficiency, sync messages should not be excessively redundant (i.e., contain many `Puts` that the recipient already knows). However, some redundancy is fine because the last-writer-wins merge rule is idempotent.

---

[6]Due to clock drift, it is possible that the "last" write is not actually the latest one. It is even possible for a write to overwrite a causally later write. However, for this assignment, you may assume clocks are synchronized well enough (even across servers) that last-writer-wins is reasonable. This is the approach taken by Apache Cassandra.

5.2   Remote Syncing

When a backend storage server first starts, it must get the existing key-value store state from older servers. To make this possible, `NewServer` receives a `remoteDescs` field that contains a description of each existing server, as returned by those servers' own `NewServer` calls.[7] Typically, each server's description will be JSON-encoded content describing how to contact that server, e.g., relevant `actor.ActorRef`s. Upon receiving existing state from an older server, the new server should merge it into all of its query actors' states using the usual last-writer-wins rule.

Once a server is aware of another server (even before completing the start-up exchange), it should immediately begin syncing new `Puts` to that server. Remote syncing is semantically the same as local syncing: updates from each actor should periodically be synced to all remote actors, using a last-writer-wins rule to merge conflicting `Puts`. However, we impose slightly different performance requirements, which may require a different implementation strategy:

- `Puts` on one actor should be visible to `Get` and `List` queries on other remote actors within **2000 ms** whenever one-way network latency is ≤ 250 ms.

- To preserve network bandwidth, you should generally avoid sending the same `Put` separately to multiple actors on the same remote server.

- To amortize overhead, for each ordered pair of actors (A, B), A should send **no more than 10 messages per second** to B, regardless of how many `Puts` A receives.

- For efficiency, sync messages should not be excessively redundant (i.e., contain many `Puts` that the recipient already knows). However, some redundancy is fine because the last-writer-wins merge rule is idempotent.

You may find it useful to define additional actor types besides `queryActor`.

For simplicity, **you do not need to handle remote message failures.** If there is an error while sending a remote message, `ActorSystem` will merely drop it (at-most-once semantics). You do not need to worry about consistency violations as a result of such a failure.[8]

---

[7]You may assume that servers are started in a strict serial order: it is never the case that two servers will be started concurrently (thus missing each others' descriptions).

[8]In particular, if actor A succeeds in sending an update to B but fails to send it to C, then B and C will be stuck in an inconsistent state, at least until a future `Put` overwrites both versions. A complete system would need to fix this consistency, e.g., by retrying the send $A \rightarrow C$ or forwarding the value $B \rightarrow C$. However, you do not need to worry about such scenarios.

6   App and Storage Client

Users play CMUD using the CMUD app, shown in Figure 4. It provides a text interface that users use to interact with the game world, including other players. The README.md file has instructions on how to play the game yourself.

Figure 4: CMUD's welcome screen.

Internally, the CMUD client (package `app`) is stateless; it gets all of its state from the backend key-value store. This makes it trivially multiplayer. Thanks to the backend store's geographic distribution and high availability, all users can get low latency by connecting to a nearby server, so it's not necessary to cache data client-side.

The `app` package interacts with the backend using a `*kvclient.Client` from the `kvclient` package. While the `app` package is implemented for you in the starter code, you will need to finish the `kvclient/client.go` file as described below.

6.1   KVS Client RPCs

A `*kvclient.Client` communicates with the backend storage system using RPCs. It sends queries to a `QueryReceiver` (defined in `kvcommon/rpc_types.go`) corresponding to a specific query actor. The client should **not** use actors or message the server's `ActorSystem` directly; instead, it should only interact with the RPC servers described in Section 3.1.

You will need to finish the `Client` implementation in `kvclient/client.go`. Specifically,

fill in the functions `Get`, `Put`, and `List` so that they make RPC calls to the corresponding `QueryReceiver` functions. The functions should be thread-safe and allow multiple outstanding queries simultaneously.

To simulate DNS load balancing, you are provided with `client.router`. Before each RPC call, call `client.router.NextAddr()` to learn which RPC server to query, in the form of an address. Note that different calls may return different addresses, to simulate the effects of load balancing (both across servers and across actors within servers), server failures, and user movement.

7   Guidelines

7.1   Starter Code

The starter code for this project can be found in the `p3/src/github.com/cmu440/` directory. You will need to implement parts of the following packages:

- In the `actor` package, files `actor/mailbox.go` and `actor/remote_tell.go` (Section 4.2).

- The `kvserver` package. Its starter code contains the API and a skeletal implementation of the `Server` you will implement.

- The `kvclient` package. Its starter code contains the API and a skeletal implementation of the `Client` you will implement.

Packages `srunner`, `crunner`, `app`, and `example` implement command-line programs that run a backend server, a `*kvclient.Client` with a simple command-line interface, the CMUD app, and an example actor program. They have been provided for your own use and do not need to be modified.

See the README.md file for a more detailed discussion of the provided starter code and tests.

7.2   Testing

Test scripts for both the checkpoint and final submission are located in the `src/github.com/cmu440/tests` directory. Please consult README.md for more details.

These test scripts are mocks of the ones we will use on Gradescope; for this assignment, there are **no** hidden tests.

7.3   Submission

Please disable or remove **all** debug prints before submitting to Gradescope. This helps avoid inadvertent failures, messy autograder outputs, and style point deductions.

For both the checkpoint and the final submission, create `handin.zip` using the following command under the `p3/` directory, and then upload it to Gradescope.

```
sh make_submit.sh
```

7.4   Advice

- Review the relevant starter code, API docs, and tools (described in the README.md) before starting on a task.

- Consult the example actor program (folder `example/`) as needed. You are free to copy and modify it.

- Keep in mind the restrictions on marshallable messages (Section 4.4) and Go's rules around methods made available for remote access.

- Before implementing local and remote syncing, plan out the actors you will use and the messages they will send. For inspiration, consider reading about Anna KVS's sync strategy [5].

8   CheckPoint (25%)

For the checkpoint, your client, server, and actor system only need to support a single backend storage server running a single query actor. This simplification removes the need for syncing (both local and remote) and the need for remote tells (item 2 in Section 4.2).

9   Final Submission

For the final submission, finish implementing all features described above. This includes remote tells, local syncing, and remote syncing. Note that the `TestOneActor*` tests from the checkpoint are re-run for the final submission.

The following table shows the point distribution for the project as a whole:

Checkpoint	21 points
Final	58 points
Manual Grading of Final	4 points
go fmt of Final	1 point
Total	84 points

Manual grading follows the same Rubric as previous projects.

10   Project Requirements

As you write code for this project, also keep in mind the following requirements:

- You must work on this project with only your partner. You are free to discuss high-level design issues with other people in the class, but every aspect of your implementation must be entirely you and your partner's own work.

- You must format your code using `go fmt` and must follow Go's standard naming conventions. See the Formatting and Names sections of Effective Go for details.

- You may use any of the synchronization primitives in Go's `sync` package for this project. Buffered channels are also permitted so long as the buffer size does not put an artificial limit on anything (e.g., the number of messages that an actor's mailbox can buffer).

- You may use any built-in Go package.

- An `Actor` implementation may not coordinate with other parts of the distributed system in any way besides receiving and sending messages. In particular, actors may not access shared memory, use Go channels, or spawn goroutines. The `actor` package will try to stop you from violating this, but we will deduct points in manual grading if you circumvent it (e.g., by mutating global variables in an actor).

- Remember to document your sync strategy (local & remote) with a 1-2 paragraph comment in `kvserver/query_actor.go`.

References

[1] Apache Cassandra, 2022. https://cassandra.apache.org/doc/latest/index.html.

[2] Ali Davoudian, Liu Chen, and Mengchi Liu. A survey on NoSQL stores. *ACM Comput. Surv.*, 51(2), apr 2018.

[3] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, oct 2007.

[4] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual consistency: Stronger properties for low-latency geo-replicated storage. *Queue*, 12(3):30–45, mar 2014.

[5] Chenggang Wu, Jose M. Faleiro, Yihan Lin, and Joseph M. Hellerstein. Anna: A KVS for any scale. *IEEE Transactions on Knowledge and Data Engineering*, 33(2):344–358, 2021.